Robot Learning

Coursework 1

Released on Wednesday 24th January 2024

Submission deadline: **Tuesday 6th February at 11pm**

Written by Edward Johns

Coursework 1 covers Lectures 1, 2, and 3. In order to complete Coursework 1, you will find it helpful to have completed Tutorials 1, 2, and 3. There is solution code to Tutorial 1 on Scientia, but the solution code for Tutorial 2 will not be released until after the deadline for Coursework 1. This is because Coursework 1 requires you to implement and evaluate your own solution.

This coursework contains six parts, and each part requires you to provide a figure, and answers to two questions. For the document you submit, you should use the Word or Latex templates which you can download from Scientia by clicking on "Data file". You should save this document as *coursework_1.pdf* when you submit it.

In the template provided, the figure placeholders show roughly what size the figures should be, and the "Lore ipsum" shows roughly how much text is required (full marks can often be obtained with just one or two sentences). You should not modify the margins or font size in the template. Each part should occupy no more than 1 page. Please add your CID number to the cover page.

For each part of Coursework 1, the figure is worth 1 mark, and each of the two answers is worth 1 mark. Coursework 1 has six parts in total, and so the total number of marks you can achieve for this coursework is 18.

When marking the figures, we will be looking for evidence that your implementation is generally correct. You should add axis titles and the values of any salient points on the axes, but the specific layout, colours, fonts etc., are not important. Figures may vary slightly across different students, due to slightly different implementations, different parameters, and the presence of random numbers in the code.

When marking the answers, we will be looking for evidence that you understand what the correct answer is, and that you are able to write this clearly and concisely. ½ a mark will be deducted if there are any clearly incorrect statements; this is to prevent students flooding the answers with guesses in the hope that one of these guesses is the correct answer.

Part 1: Random Shooting

Complete Part 1 of Tutorial 2. Set *configuration.ENVIRONMENT_TYPE* to 'medium', set *configuration.PLANNING_METHOD* to 'random shooting', and set *configuration.SHOOTING_NUM_PATHS* to 100. For the reward function, use the Euclidean distance between the state at the end of the path, and the goal.

Figure 1: This figure explores the relationship between the path length, and the reward for the best path. Create a line graph, where the y-axis is reward of the best path, and the x-axis is the path length (*configuration.SHOOTING_PATH_LENGTH*). Data points should be plotted for the following values for the path length: [10, 20, 50, 100, 200, 500, 1000]. For each of these values on the x-axis, the y-axis value should be the average value calculated across 5 runs of the full random shooting algorithm. Points should be joined together with a straight line.

Question 1a: Inspect the shape of the line graph, and consider if it is a straight line, a curve with a peak, or a curve which saturates. Explain why the line graph has this shape.

Question 1b: Instead of the reward function being the Euclidean distance between the path's final state and the goal, an alternative reward function would be to multiply this Euclidean distance by 2. What effect, if any, would this have on the performance of the random shooting algorithm?

Part 2: Cross-Entropy Method

Complete Part 2 of Tutorial 2. Set *configuration.ENVIRONMENT_TYPE* to 'medium', set *configuration.PLANNING_METHOD* to 'cross-entropy method', set the number of paths (e.g. *configuration.CEM_NUM_PATHS*) to 30, set the path length (e.g. *configuration.CEM_PATH_LENGTH)* to 50, and set the number of elites (e.g. *configuration.CEM_NUM_ELITES*) to 5 (this is the value of k in the algorithm presented in Lecture 2). For the reward function, use the Euclidean distance between the state at the end of the path, and the goal.

Figure 2: This figure explores the relationship between the distance to the goal for the best path, and the number of iterations of the cross-entropy method. Create a line graph, where the x-axis is the number of iterations of the cross-entropy method, and the y-axis is the Euclidean distance between the goal and the state at the end of the best path sampled in the final iteration. Data points should be plotted for the following values on the x-axis: [1, 2, 3, 4, 5]. For each of these values on the x-axis, the y-axis value should be the average value calculated across 5 runs of the full cross-entropy method algorithm. Points should be joined together with a straight line.

Question 2a: If this planning algorithm were to have an infinite number of iterations, is it guaranteed, for each new run of the algorithm, that the robot would plan a path that reaches close to the goal? Explain your answer.

Question 2b: Let's say that in the final iteration, the best path reaches the goal perfectly, without hitting the obstacle. Once the final mean action sequence has been calculated, is it guaranteed that the robot would avoid hitting the obstacle when it executes this action sequence?

Part 3: Gradient-Based Planning

Complete Part 3 of Tutorial 2. If you are finding this difficult, I will be providing some help in Lecture 3. Set *configuration.ENVIRONMENT_TYPE* to 'medium', set *configuration.PLANNING_METHOD* to 'gradient-based', set the number of waypoints (e.g. *configuration.GRAD_NUM_WAYPOINTS*) to 20, and set the number of iterations (e.g. *configuration.GRAD_NUM_ITERATIONS) to 100.*

Figure 3: This figure visualises the waypoints after the optimisation is complete. Once your gradient-based planning is working well, you should create a figure showing what this path of waypoints looks like after the waypoints have been successfully optimised to allow the robot to reach the goal. This figure should show the right-hand half of the window that is displayed, i.e. the "Planning Visualisation" half.

To save an image of the window displayed, you can use the following code:

```
buffer = pyglet.image.get_buffer_manager().get_color_buffer()
image_data = buffer.get_image_data()
image_data.save('screenshot.png')
```

(You may also wish to simply capture a screenshot.)

Question 3a: Why it is important to include a smoothness term in this reward function?

Question 3b: In this implementation of gradient-based planning, where a path of waypoints is optimised, does the robot's model of the dynamics need to be differentiable? Explain your answer.

Part 4: Model Learning

Complete Part 1 of Tutorial 3. Set *configuration.ENVIRONMENT_TYPE* to 'hard', set *configuration.PLANNING_METHOD* to 'random exploration', and set *configuration.RANDOM_PATH_LENGTH* to 100.

Figure 4: This figure visualises the model learned after a period of random exploration. Collect 3 episodes of random exploration data (each with a path length of 100), and then train the model using all of this data (until the model has converged). Then, create a figure showing the "Model Visualisation" section of the window, to show what the model has learned. The model should show the predicted next state for when the action is [0.3 * *constants.ROBOT_MAX_ACTION*, -0.8 * *constants.ROBOT_MAX_ACTION*]. (See Part 3 above for how to save an image of the displayed window, or you may simply capture a screenshot).

Question 4a: Why is the learned model more accurate in the bottom part of the environment, than in the top part of the environment?

Question 4b: In this environment, is the true environment dynamics a linear function or a non-linear function? Explain your answer.

Part 5: Model-Based Reinforcement Learning

Complete Part 2 of Tutorial 3, and try to develop an algorithm that will enable the robot to reach the goal (or close to the goal) using the model it has learned. Set *configuration.ENVIRONMENT_TYPE* to 'hard', set *configuration.PLANNING_METHOD* to 'cross-entropy method', and set *configuration.CEM_PATH_LENGTH* to 50.You should first allow the robot to collect 3 episodes of random exploration data with a path length of 100 (as in Part 4 above), and then you should train the model on this data (until the model has converged), before then using the cross-entropy method for all subsequent planning with a path length of 50. Remember that when the robot executes the planned actions, it should collect the data it observes and further train its model.

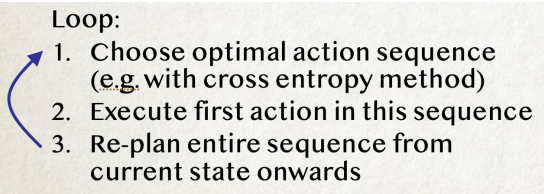
Figure 5: This figure visualises how close the robot gets to the goal after each episode it executes in the environment. Create a line graph, where the x-axis is the number of episodes the robot has executed in the environment ("physical" episodes in the real environment, not episodes during the planning), and the y-axis is the Euclidean distance between the goal and the state at the end of that episode. The values on the x-axis should be [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. For the x-axis values of 1, 2, and 3, the y-axis value should be for each of the initial 3 episodes of random exploration. For the x-axis values of 4, … 10, the y-axis value should be for the subsequent episodes planned using the cross-entropy method.

Question 5a: Why is it difficult for the robot to reach the goal when planning after training the model only on the initial 3 episodes of random exploration?

Question 5b: In model-based reinforcement learning, the robot usually trains its model on the (*state*, *action*, *next state*) data it collects when it moves through the physical environment. Would there be any advantage or disadvantage of training on the (*state*, *action*, *next state*) data from the paths sampled during the cross-entropy method planning?

Part 6: Model-Predictive Control

Currently, you have implemented "open-loop planning". You should now implement "closed-loop planning" (model-predictive control), according to the algorithm presented in Lecture 3:



This should be integrated into the code you have already written for Part 5 above. The code for the model learning should remain the same, and the only change should be that when the robot moves through the physical environment, it re-plans its path after each action it executes. The robot should always be planning 50 actions ahead, and the episode should end once the robot has executed 50 actions in the physical environment.

Figure 6: This figure visualises the model learned after running the full algorithm. After implementing the full model-based reinforcement learning algorithm with model-predictive control, create a figure showing the "Model Visualisation" section of the window, to show what the model has learned. The model should show the predicted next state for when the action is [-0.8 * *constants.ROBOT_MAX_ACTION*, 0.1 * *constants.ROBOT_MAX_ACTION*

]. (See Part 3 above for how to save an image of the displayed window, or you may simply capture a screenshot).

Question 6a: After introducing model-predictive control, is it guaranteed that the robot would avoid hitting the obstacle when attempting to reach the goal?

Question 6b: If the robot has access to the true environment dynamics (or if its model is learned perfectly), would there be any benefit in introducing model-predictive control?

End of Coursework 1